ISI Research Report
ISI/RR-88-226
June 1989

(4)

University of Southern California

K. M. Chandy
R. Sherman

The Conditional–Event Approach to Distributed Simulation

INFORMATION SCIENCES INSTITUTE
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT This document is approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ISI/RR-88-226	5. MONITORING ORGANIZATION REPORT NUMBER(S) -------
6a. NAME OF PERFORMING ORGANIZATION USC/Information Sciences Institute 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 4676 Admiralty Way Marina del Rey, CA 90292-6695	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Street Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA-Ames Strategic Defense Initiative Organization 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NCC 2-539 N00014-87-K-0022
8c. ADDRESS (City, State, and ZIP Code) NASA-Ames Moffett Field, CA 94035 SDIO Office of the Secretary of Defense The Pentagon Washington, DC 20301	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. ------- PROJECT NO. ------- TASK NO. ------- WORK UNIT ACCESSION NO. -------

11. TITLE (Include Security Classification)

The Conditional-Event Approach to Distributed Simulation (Unclassified)

12. PERSONAL AUTHOR(S) Chandy, K. M.; Sherman, R.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Research Report	FROM _____ TO _____	1989, June	27

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	distributed simulation, conditional event
09	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report proposes a new conservative approach to distributed simulation that appears to be efficient. Performance measures of the simulator executing queueing network models are discussed. This work departs from earlier work on distributed simulation in two aspects. The topological structure of simulators described in the literature mirrors the structure of the physical system. By contrast, in our method, messages are used for a variety of purposes, and the process interconnection structure of the simulator need not reflect the structure of the physical system. The second difference between the method proposed here and that used in the literature is our use of conditional events, an extension of the concept that forms the basis for sequential simulation. Experimental results on the Intel iPSC/1 running Cosmic C are reported.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Victor Brown Sheila Coyazo	22b. TELEPHONE (Include Area Code) 213/822-1511 22c. OFFICE SYMBOL

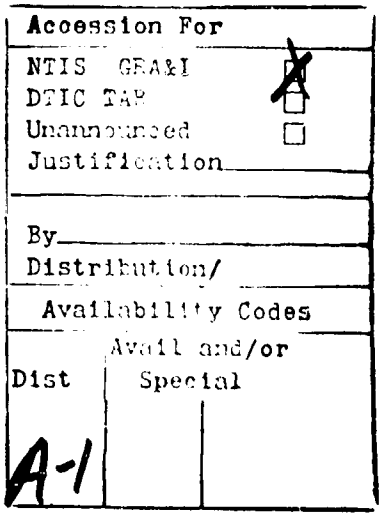
DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ISI Research Report
ISI/RR-88-226
June 1989

K. M. Chandy
R. Sherman

The Conditional–Event Approach to Distributed Simulation



INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

This research is supported by the Strategic Defense Initiative Office under Office of Naval Research Contract No. N00014-87-K-0022, and by NASA-Ames under Cooperative Agreement NCC 2-539. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of the Strategic Defense Initiative Office, the Office of Naval Research, NASA-Ames, or the U. S. Government. No official Government endorsement should be inferred.

Acknowledgments

We thank David Mizell for his continuous support, and Susan Coatney for her valuable help with the implementation.

A shorter version of this report was published as Chandy, K. M., and R. Sherman, "The Conditional Event Approach to Distributed Simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation,* pp. 95-99, Tampa, Florida, March 1989.

The Conditional-Event Approach to Distributed Simulation

K. M. Chandy
California Institute of Technology*

R. Sherman
USC/Information Sciences Institute

August 30, 1988

1 Introduction

1.1 Approaches to Distributed Simulation

One of the advantages of computers is that they can be used to simulate systems that cannot be observed directly. Parallel computers are used extensively for simulation. Many parallel simulators are *time-driven*: At the k-th step, where $k \geq 0$, the simulator computes the state at time k of all processes in the physical system (i.e., the system being simulated). This paper is concerned with *event-driven* distributed simulation in which the simulator may compute the behavior of different subsystems up to different points in time. There are two common approaches to event-driven distributed simulation: the conservative approach [1, 7] and the optimistic approach [6]. A computation in the conservative approach proceeds by determining the correct behavior of each physical process in a time interval. A computation in the optimistic approach may compute an incorrect behavior that is corrected later by rollback and recovery. This paper presents a new conservative approach that appears to be efficient. Performance measurements of the simulator written in Cosmic C, running on an Intel iPSC/1, are provided.

*On leave of absence from The University of Texas at Austin.

1.2 Our Point of Departure

Most of the distributed simulators described in the literature are designed so that the structure of the simulator mirrors the structure of the physical system: A message between a pair of processes in the simulator represents an event that changes the states of the corresponding pair of processes in the physical system. By contrast, in our method messages are used for a variety of purposes, and the process-interconnection structure of the simulator need not represent the structure of the physical system. There is no reason to constrain the structure of the simulator program to reflect the physical world. Our goal is to exploit the architecture of a parallel computer to obtain fast execution, and the relevant yardstick is execution time. At an abstract level, distributed simulation algorithms are suitable for all parallel architectures, but there are differences at the level of detailed implementation that impact performance. The efficiency of a simulation program depends on the characteristics of the target computer: the underlying architecture (message-passing multicomputer, shared-memory multiprocessor [11]), the amount of memory, the number of processors, the speed of synchronization, and process switch time. The question of interest is how appropriate a simulation algorithm is for a given application and a given target computer.

Another difference between the method proposed here and those in the literature is our use of *conditional events*. The event list in a sequential simulation is a list of conditional events: The meaning of an event in the event list is that it is the next event to be executed *provided there is no earlier event*. For example, a preemptive priority queue serving a low-priority customer with a remaining service time of T units will complete service of the customer T units later, *provided no higher priority jobs arrive while the low-priority customer is being served*. The earliest event in the event list is the next event to be executed, even though it is a conditional event. Thus the event list is a mechanism for determining definite events (i.e., events that occur in the system being simulated) from conditional events. We employ conditional events in much the same way that they are employed in sequential simulation. We also use other ways of determining definite events. Consider again a preemptive priority queue serving a highest priority customer with a remaining service time of T units. This customer will depart after T time units no matter what happens in the future. Thus we can determine that the departure of the highest priority customer is a definite event without using the event list. A critical element of the success of our method is determining as many definite events as possible without using

the fact that the earliest conditional event is also a definite event. The use of conditional events guarantees absence of deadlocks, and furthermore we need not use *null* messages to guarantee progress [7].

2 The Physical System

2.1 Physical Processes

Our goal is to simulate a system on a parallel computer. The system that is to be simulated is called the *physical* system, and a process in the physical system is called a *physical process* (or PP) in contrast to a process in the simulator, which is called a *logical process* (or LP).

In the physical system, time is integer-valued. Initially time has value 0. We are required to determine the behavior of the physical system at all times in the interval $[0, H]$ where H is the horizon of the simulation.

A set of input ports and a set of output ports are associated with each PP; inputs to the PP are received along its input ports, and outputs from the PP are sent along its output ports. The state and the outputs of a PP at time $t+1$ are functions of its state and its inputs at time t for all t where $t \geq 0$. Thus a PP is defined by a set of input ports, a set of output ports, a set of states, an initial state and a next-state function, and an initial output and a next-output function for each output port.

2.2 A System of Physical Processes

A physical system is a set of PPs and a set of *connections* where each connection connects one output port of a PP to one input port of another PP. The value of an input port is (always) equal to the value of the output port that it is connected to. The state of a physical system is given by the states of its PPs and the values of the ports of its PPs. Thus the state of a physical system at time $t+1$ is a function of its state at time t, for all t where $t \geq 0$. The initial system state is determined by the initial states of its component PPs. The problem is to compute the state of the physical system for all times t in the interval $[0, H]$.

A simple solution is to compute the state of the physical system for increasing values of t, for all t in $[0, H]$. In this paper we design a parallel solution.

All PPs and all ports have unique names.

3 The Simulator

3.1 Overview of the Algorithm

The simulator is described for a message-passing multicomputer implementation in which messages are delivered in the order sent. We first describe a synchronous version of the algorithm and later describe an asynchronous version.

The simulator has one logical process (LP) for each PP. Let PP X be simulated by LP X, for all X. Assume, for the time being, that the simulator network is fully connected, i.e., each LP can send messages to all LPs.

The basic operation of LP X is a repetition of the following loop:
begin-loop

1. Determine inputs to PP X from initial conditions and messages it has received from other LPs.
2. Compute outputs from PP X as far into the future as possible given the inputs to PP X that have been determined so far, and send a message to each LP.
3. Wait to receive a message from each LP.

end-loop

The algorithm is synchronous in the sense that an LP receives messages from all other LPs on each execution of the loop. The synchronous algorithm is simpler than the asynchronous version in which an LP does not wait to receive a message from every other LP, and so we describe the synchronous version first.

3.2 Local Variables of an LP

An LP X has the following local (integer) variables:

- a time u_r for each input port r
- a time u_s for each output port s
- a set E_X of pairs (op, msg) where op is an output port of PP X and msg is a message sent along op
- a time $next_X$

- a time $C_X[Y]$ for each LP Y

The local variables of LP X have subscript X, r, or s; this makes it easier to describe the algorithm because each variable used in the algorithm has a distinct name (because all processes and ports have unique names).

Local variables u_r and u_s have the following meaning:

- LP X has received messages from another LP defining the input to PP X along port r in the interval $[0, u_r]$, and
- LP X has sent messages to another LP defining the output from PP X along port s in the interval $[0, u_s]$.

Variable $C_X[Y]$ is a buffer to store the $next_Y$ field of messages received from LP Y; all messages from LP Y contain a $next_Y$ field, and when LP X receives a message from LP Y it stores the value of the $next_Y$ field of the message in $C_X[Y]$. An invariant of the program is that if there are no messages in transit from LP Y to LP X then $C_X[Y] = next_Y$. Similarly, u_r may be thought of as a buffer for u_s, where ports r and s are connected. Let s be an output port of PP Y connected to input port r of PP X. Every message sent by LP Y to LP X contains the value of u_s, and when LP X receives the message it stores this value in u_r. An invariant of the program is that if there are no messages in transit from LP Y to LP X then $u_r = u_s$. Variables u_r and u_s are monotone nondecreasing.

The Earliest Conditional Event. Let $cond_s$ be the time at which the next message is sent by PP X along output port s after time u_s *if there are no inputs along* r *after time* u_r, for all input ports r of PP X. Define $next_X$ as the minimum of $cond_s$ over all output ports s of PP X. Define E_X as the set of pairs (op, msg) where PP X sends message msg along output port op at time $next_X$ *if there are no inputs along* r *after time* u_r, for all input ports r of PP X.

Example. Consider a preemptive priority queue with one input port r through which all customers arrive, and one output port s through which all customers depart. We treat the queue as a PP and customers as messages. Since there is only one output port, $next_X = cond_s$. Initially $u_r = 0$ and $u_s = 0$. Customers in the queue have one of two priorities: high or low. We shall show that if initially the customer at the head of the queue has a

service time of T units, then $cond_s = T$, and if initially the queue is empty then $cond_s = \infty$.

Suppose the customer at the head of the queue is a high-priority customer. Then this customer will leave the queue at time T, no matter what future inputs are. Therefore, in this case, $cond_s = T$, and E_X is $\{(s, HI)\}$ where HI represents the departure of a high-priority customer.

Now suppose the customer at the head of the queue is a low-priority customer (and in this case there are no high-priority customers in the queue initially). If a high-priority customer arrives before T, then the low-priority customer is preempted and must wait at least until the arriving high-priority customer finishes service. If no high-priority customers arrive at the queue before T, then the low-priority customer will leave the queue at time T. Therefore, $cond_s = T$ because $cond_s$ is the time of the next departure after time u_s along port s *if there are no arrivals along port* r *after time* u_r. In this case, E_X is $\{(s, LO)\}$ where LO represents the departure of a low-priority customer.

Finally, consider the case where the queue is empty initially. No customer leaves the queue if no customer enters the queue, and hence $cond_s = \infty$. In this case, E_X is $\{(s, null)\}$ where *null* represents the departure of no customers.

The meaning of $cond_s$ is an extension of the meaning of the time of a conditional event in sequential simulation.

In addition to the variables described here, an LP has other local variables to carry out the simulation of the corresponding PP and to gather statistics; at this time we choose to ignore these variables and to focus attention on the variables required for communication.

Note: The symbol u stands for *u*pto: Communication along a port r has been computed *u*pto time u_r. The symbol s is used for output ports because it stands for *s*ending port; similarly, the symbol r is used for input ports because it stands for *r*eceiving port.

3.3 Messages Exchanged by LPs

A message sent by an LP X to an LP Y contains the following information:

- For each output port s of PP X that is connected to an input port of PP Y: A pair (s, D_s) where D_s describes the output along port s of PP X after time u_s.

- The time $next_X$ defined earlier.

We now define D_s.

Representing Physical Communication. A sequence of outputs after time u_s along an output port s of a PP is represented in the simulator by the sequence D_s of pairs $(t[i], e[i])$, $0 \leq i \leq K$ where $K \geq 0$, $t[i]$ is a time where $t[i] > u_s$, and $e[i]$ is either an output along s or the special symbol *null*. Sequence D_s is ordered in increasing order of $t[i]$ and represents the outputs along s in the interval $(u_s, t[K]]$ in a simulation in which:

- $e[i]$ is output along s at time $t[i]$ if $e[i]$ is not *null*, and nothing is output along s at $t[i]$ if $e[i]$ is *null*; and
- nothing is output along s in the interval $(u_s, t[K]]$ at times other than $t[i]$, $0 \leq i \leq K$.

Concurrent with sending the message containing D_s, u_s is assigned the value $t[K]$; and when an LP receives a message containing D_s, u_r is assigned the value $t[K]$ where r is the input port connected to s. Therefore, u_s and u_r are monotone nondecreasing.

Example. Consider a simulation in which for some output port s of a PP X the following values are output after time 3: B at time 5, C at time 10, B again at time 20; and there are no other outputs along s in the interval (3, 30]. With $u_s = 3$, the outputs along port s in the interval (3, 30] can be represented in the simulation by a message containing the following sequence: (5,B), (10,C), (20,B), (30,*null*). Concurrent with the sending of the message, u_s becomes 30; and when an LP receives the message, it sets u_r to 30.

3.4 The Algorithm

Initially, for all input ports r: $u_r = 0$, and for all output ports s: $u_s = 0$. For simplicity, assume that no messages are sent in the physical system at time 0; therefore, assume that initially all LPs have received messages $(0, null)$ for each input port and have received messages with $next_Y = 0$ from all LPs Y. Hence, initially $next_X = 0$, $C_X[Y] = 0$, and $E_X = \{(s, null)\}$ where s is an output port of PP X.

An LP X repeatedly executes the following loop:

begin-loop

1. **Obtain definite events from conditional events**
   If equation 1 (below) holds,

$$next_X = minimum\ over\ all\ Y\ of\ C_X[Y]. \tag{1}$$

   then msg is (definitely) output by PP X along port op at time $next_X$ for all pairs (op, msg) in E_X.
   Compute as many definite events as possible
   The messages received by LP X describe the input to PP X along port r upto time u_r, for all r. Use the initial conditions, the messages received, and the definite events obtained from conditional events to compute the (definite) output of PP X as far into the future as possible.

2. **Message sending**
   Update $next_X$ and E_X and send a message to each LP Y — the message contains the (updated) value of $next_X$, and for each output port s of PP X connected to an input port of PP Y the value (s, D_s) — and update u_s.

3. **Message receiving**
   Wait to receive a message from each LP. Upon receiving a message from LP Y, set $C_X[Y]$ to the $next_Y$ field of the message, and update u_r for all input ports r of PP X connected to PP Y.

end-loop

Example. Consider a cyclic queueing network with three preemptive priority queues: q_0, q_1, q_2, where the outputs of q_0, q_1, q_2 are the inputs to q_1, q_2, q_0, respectively. There are two priorities of customers — high and low — and priorities do not change. The system contains three customers who remain in the system forever; customers do not enter or leave. Initially q_0 contains a high-priority customer with a service time of 13 units, q_1 contains a low-priority customer with a service time of 17 units, and q_2 contains a low-priority customer with a service time of 11 units. For simplicity, assume that a customer's service time is the same in all queues. Let ri and si be the input and output ports (respectively) of queue i, $i = 0, 1, 2$. The first few iterations of the simulation are given in Table 1.

The table shows the values of variables at the *end* of each iteration. Now we shall discuss the first iteration in detail.

iteration		0	1	2	3
queue 0	next	0	13	∞	24
	E	r ıll	HI	null	LO
	u - r0	0	0	11	11
	u - s0	0	13	13	13
	D		13,HI	()	()
queue 1	r∘xt	0	17	26	30
	E	null	LO	HI	LO
	u - r1	0	13	13	13
	u - s1	0	0	26	26
	D		()	26,HI	()
queue 2	next	0	11	11	39
	E	null	LO	LO	HI
	u - r2	0	0	0	26
	u - s2	0	0	11	39
	D		()	11,LO	39,HI

Table 1: A simulation

A high-priority customer will leave queue 0 at time 13 no matter what future inputs may be. Therefore, LP 0 sends a message with $D_{s0} = (13, HI)$ to LP 1 and concurrently sets u_{s0} to 13. Since a high-priority job leaves at time 13, $next_0 = 13$ and $E_0 = \{(s0, HI)\}$. Since LP 0 receives a message from LP 2 with D_{s2} being the empty sequence, u_{r0} remains unchanged at 0.

Now consider queue 1. A low-priority customer will depart the queue at time 17 *if* no high-priority customer arrives earlier. Therefore, $next_1 = 17$ and $E_1 = \{(s1, LO)\}$. LP 1 sends a message with $D_{s1} = ()$ to LP 2, and leaves u_{s1} unchanged because no definite output from queue 1 can be predicted. The behavior of queue 2 is similar.

On the second iteration, LP 2 converts the conditional event of a departure of a low-priority job at time 11 into a definite event because it has the smallest value of *next*. The remainder of the table is straightforward.

3.5 Outline of Proof of Correctness

Safety. We are required to prove the invariant that all outputs of LPs are outputs of PPs. For our program this reduces to showing that the

conversion of conditional events to definite events is correct. We shall show that if $next_X$ = minimum over all Y of $C_X[Y]$, then the outputs in E_X are sent at time $next_X$ in PP X.

We shall use the superscript k to denote the value of a variable at the *end* of the k-th iteration. From the program:

$$C_X^k[Y] = next_Y^k. \tag{2}$$

$$u_r^k = u_s^k, \text{ for all ports } r, s \text{ connected to each other.} \tag{3}$$

Define T as follows:

$$T = \text{minimum over all } Y \text{ of } C_X^k[Y]. \tag{4}$$

From equation 2 and the above:

$$T = \text{minimum over all } Y \text{ of } next_Y^k. \tag{5}$$

We shall show by induction on t that for all t where $t < T$, there is no output (in the physical system) along port s in the interval $(u_s^k, t]$ for all output ports s in the system, and there is no input along port r in the interval $(u_r^k, t]$ for all input ports r in the system.

Base Case. The induction hypothesis holds vacuously for $t = 0$.

Induction Step. Assume the hypothesis true for all times up to and including $t-1$, where $t < T$, and we shall prove it true for t. For all input ports r there is no input in the interval $(u_r^k, t-1]$, from the induction hypothesis. From the meaning of $next_Y^k$, for all t where $u_s^k < t < next_Y^k$, there is no output along port s of PP Y at time t *if* for all input ports r of PP Y there is no input along port r in the interval $(u_r^k, t-1]$; from equation 5 the same holds for all t where $u_s^k < t < T$. Therefore, for all output ports s there is no output at time t where $u_s^k < t < T$. Employing the induction hypothesis, there is no output along port s in the interval $(u_s^k, t]$. From equation 3 there is no output along port s in the interval $(u_r^k, t]$. Since the value of an input port is equal to the value of the output port that it is connected to, there is no input along an input port r in the interval $(u_r^k, t]$. This completes the proof by induction.

Let $next_X^k = T$. We have shown that for all input ports r of PP X there are no inputs in the interval (u_r^k, T). From the meaning of $next_X^k$ it follows that at time $next_X^k$, PP X outputs msg along port op for all (op, msg) in E_X^k.

Progress. For the k-th iteration, all k, there is at least one LP X such that:

$$next_X^k = minimum\ over\ all\ Y\ of\ next_Y^k. \tag{6}$$

This LP determines at least one definite event. This event is output on the k-th iteration if it can be determined that the event is definite without using equation 6; otherwise, it is output on the $k+1$-th iteration. Thus at least one LP outputs an event at least once in every two iterations. Since the outputs along a port are monotone increasing in simulated time, the simulation progresses.

4 Asynchronous Version

In the synchronous version, an LP receives messages from all other LPs and sends messages to all other LPs in each iteration. Now consider an asynchronous version in which there are no such iterations. An LP computes the output of the corresponding PP as far into the future as possible, and sends this output to the appropriate LPs. When an LP receives a message from *any* other LP, it computes further output of the PP and sends the output. Thus an LP need not wait to receive messages from *all* other LPs before computing further and sending messages. The simulation may not progress because all LPs may be idle, waiting to receive messages. Conditional events can be used to avoid deadlock.

As long as there are messages waiting in an LP's input buffer, the LP does not use conditional events. Conditional events are used only when the LP would be otherwise idle. The proof of the synchronous version suggests how the asynchronous program can be structured. The proof rests on equations 2 and 3, and these equations play a key role in the design of the asynchronous algorithm. Our goal is to design an algorithm in which an LP X can determine whether equations 2 and 3 (or equivalent equations) hold. We now present an algorithm and later discuss its proof.

The Algorithm. Each LP Y records $next_Y$ and, for each of its input ports r, the number of messages n_r it has received along r, and, for each of its output ports s, the number of messages n_s it has sent along s. The recording must be an atomic action in the sense that the values of $next_Y$, n_r, and n_s must not change during the recording. The recording is carried out at *arbitrary times* with the constraint that an LP re-records its values some finite time after they change. LP Y broadcasts (also at arbitrary times) the

values it has recorded, again with the proviso that it re-broadcasts values some finite time after they change. An LP X has local variables $C_X[Y]$, $D_X[r]$, and $D_X[s]$, in which it retains the last values received of $next_Y$, n_r, and n_s, respectively, for all LPs Y where $Y \neq X$, and all ports r, s of PPs in the system other than PP X itself. LP X guarantees that $C_X[X] = next_X$, and for all its input ports r, $D_X[r] = n_r$, and for all its output ports s, $D_X[s] = n_s$. (Therefore, LP X does not send messages to itself.)

Conditional events are converted to definite events as follows. We are given the conditional event that PP X outputs the messages in E_X at time $next_X$ *if* for each of its input ports r, it receives no message along r after time u_r. If equation 1 and equation 7 (below) hold, then PP X (definitely) outputs the messages in E_X at time $next_X$.

$$\textit{for all ports } r, s \textit{ connected to each other} : D_X[r] = D_X[s]. \qquad (7)$$

Note that all the variables named in equations 1 and 7 are local to LP X.

Outline of Proof. The proof is based on the concept of global snapshots [2, 3]. A global snapshot is a state of the system (and in this case the system is the simulator) that *could* have occurred earlier. If each process records its local state, the messages it has sent on each output port, and the messages it has received on each input port, then *the collection of local recordings is a global state if and only if the number of messages sent along each output port is greater than or equal to the number of messages received along the input port that it is connected to.* (We assume that initially there are no messages in transit.) The state of a channel from an output port s to an input port r, in the global state, is the sequence of messages sent along s in the recording, excluding the sequence of messages received along r in the recording.

If equation 7 holds, then the values $next_Y = C_X[Y]$, $n_r = D_X[r]$, and $n_s = D_X[s]$ form (part of) a global state G, because for all connected pairs of ports s and r, the number of messages sent along port s in G is equal to the number of messages received along r in G. In global state G, $next_Y = C_X[Y]$; see the similarity to equation 2. Also, for all connected ports r and s, $u_r = u_s$ in G — see the similarity to equation 3 — because $n_r = n_s$ in G. Now the proof that PP X outputs the messages in E_X at time $next_X$ is exactly the same as in the synchronous version, with the values of variables at the k-th iteration in the synchronous version representing the values of the corresponding variables in global state G in the asynchronous version.

Outline of Proof of Progress. We shall show that the simulation does not reach a deadlocked state in which all LPs are waiting for messages and all channels are empty. If the system remains in a state in which there are no messages in transit, then (from the algorithm), for all LPs X, equation 7 holds (because $n_r = n_s$ since there are no messages in transit, and $D_X[r] = n_r, D_X[s] = n_s$, since LP X receives the latest recorded values from all processes). By the same argumemt, for LPs X, Y: $next_Y = C_X[Y]$. Consider the LP, say X, with the smallest value of *next*. For this LP, equation 1 holds as well, and hence it converts a conditional event to a real event. Therefore, at least one LP eventually makes progress.

5 Experimental Results

5.1 The Physical System

For our initial experiments, we chose problems that appear to be unsuited for conservative methods. Experiments in the literature [8] suggest that conservative distributed simulation methods are inefficient for queueing networks in which the routes that customers follow through networks are random. Therefore, for our first set of experiments we chose networks with several *switches*, where each switch has several incoming edges and several outgoing edges; a customer entering a switch leaves via one of the outgoing edges with a given probability. A customer leaving a switch enters a tandem sequence of queues and then enters another switch. Experiments were done both for first-come-first-served (fcfs) queues and for preemptive-priority queues. In the later case, each customer in the network is assigned a priority: either high or low. The network is closed, i.e., customers neither leave nor enter the network, and customers that are initially in the network remain in the network forever. Our experiments are characterized by the number N of switches, the number L of queues between the output of one switch and the input of another, the probabilities associated with each output edge from each switch, and the number of customers in the network. The last parameter is denoted by J in the case of fcfs queues and by J_l, J_h, standing for the number of low- and high-priority customers, respectively, in the case of preemptive-priority queues. A variety of topologies can be simulated by setting the probability of following some output edges of a switch to 0, and others to 1.

A strong argument can be made that queueing networks (or indeed any system with a large stochastic component) should not be simulated using

distributed simulation. The time taken to simulate stochastic systems is largely spent in reducing the variance of the results, and the most efficient scheme for variance reduction is replication. We use queueing networks because they serve as easily understood benchmarks.

5.2 Implementation

The synchronous version of the algorithm was implemented in *Cosmic C*[10] running on an Intel iPSC/1. We did not experiment with the asynchronous version, because the synchronous version gave almost linear speedup, and the asynchronous version cannot do much better. A few points about the implementation are of general interest, and are not limited to the example discussed here.

5.2.1 Mapping the Physical System to a Multicomputer

Process Switching. The time taken for the operating system to switch control from one process to another, in a node of a multicomputer, can be substantial. We reduce process-switching overhead by executing switching *within the simulation* rather than by calling the operating system.

Load Balancing. An important aspect of efficiency in concurrent systems is load balancing: The implementation should be designed so that all nodes in the system have about the same amout of load. We mapped the queueing network onto the computer to ensure that each node was (roughly) equally active.

Grain Size. If the execution time between message-sending is small compared to message delay, the overhead of communication can negate the speed gained from concurrent execution. In our experiments, care was taken to ensure that the the process grain size matched the computer. The networks we simulated were large, and the amount of computation per message was of the same order as message delay.

Lookahead. The simulation is written so that each LP determines the behavior of a PP as far into the future as possible. An LP ceases computation, and waits for additional input, only if it cannot determine any more outputs of the PP that it is simulating.

Caveat. The success of our experimental results is probably based on the care with which the problem was mapped onto the iPSC/1. Poor results may be obtained from a simulation in which:

- the overhead of process-switch time plays a significant role in the execution time of the simulator, or
- the loads across nodes of the computer are significantly out of balance, or
- there is a mismatch between the process grain size and the target computer, or
- LPs do not look ahead.

5.2.2 Unifying Conservative Methods

A goal of our experiments is to unify conservative methods on the one hand, and ideas that have been developed in sequential simulation on the other. We employ *null* events by which one LP informs another how far it has progressed in the simulation, and in this sense, our method is similar to *null* event schemes in the literature; but we do not depend on *null* events to guarantee progress. The speedup obtained does depend significantly on the use of *null* messages. We employ conditional events in much the same way that they are employed in sequential simulation, and in this sense, our method is an extension of sequential simulation. The use of conditional events can be thought of as one way of breaking deadlock; and in this sense, our method has some similarity to deadlock-breaking schemes, except that there is no need for a deadlock-detection algorithm.

A goal of our (continuing) work on efficient implementations of distributed simulation is a search for paradigms that unify apparently disparate ideas.

5.3 Results

Experiments were done for networks of $N = 12$ and $N = 24$ switches, with path lengths $L = 5$ and 10. For fcfs queues, we experimented with number of customers J: $N \times 10$, $N \times 50$, and $N \times 100$. In the case of priority queues, experiments were done with J_l, J_h (the number of low- and high-priority customers, respectively) as follows : $N \times (10, 10)$, $N \times (50, 10)$, and $N \times (90, 10)$.

Each of the above networks was simulated once with equal probabilities for each output port of each switch and once with different probabilities for each output port (e.g., for $N = 12$, probabilities range from 0 to 0.2). The results were very similar for the same network for the two cases (i.e., equal and different probabilities).

Speedup was computed relative to the execution time for one simulation process. For this example, the sequential version resulting from the distributed algorithm was at least as efficient as a central event queue implementation.

Figures 1 through 3 and 5 through 7 summarize the speedups computed for the fcfs networks of size $N = 12$ and $N = 24$, respectively, using $M = 2$ to 24 processors. Figures 4 and 8 give the maximal speedups computed for these networks for different values of J, for $N = 12$ and $N = 24$, respectively. Figures 9 through 11 summarize the speedups computed for the priority queuing networks with $N = 24$, and figure 12 give the maximal speedups computed for this network for the different values of J_h, J_l.

6 Conclusions

The ratio of the execution time of a concurrent program to the execution time of a sequential program (for the same problem) depends on several factors:

- **The concurrent and sequential algorithms employed**

  In our case, the conventional sequential discrete-event simulation program is *slower* than the concurrent algorithm running on a single processor. This is because the concurrent algorithm is tailored to the problem (simulating a network of priority queues) whereas the sequential algorithm is general. Therefore, in our analysis, we compared the execution times of the concurrent algorithm running on computers with different numbers of processors.

- **Computations on global states**

  In most physical systems, the behavior of a PP can affect the behavior of all other PPs. (For instance, a job departing from a queue may eventually arrive at each of the other queues.) Because of this 'global' effect, most distributed simulations carry out some computation on the (global) state of the simulation; deadlock detection and

the computation of global virtual time [6] are examples of such computations. Global computations require communication between LPs. The greater the frequency of such computations, the greater the communication overhead.

The implementation of our algorithm is semi-synchronous: the algorithm consists of repeated executions of a loop in which LPs carry out a simple global computation and then each LP carries out a local computation. The efficiency of the algorithm depends primarily on the amount of local computation in the loop: if there is a large amount of local computation, the advantage of concurrency outweighs the overhead of global computation. Therefore, the algorithm provides speedup if the problem is large (i.e., it requires a great deal of computation), the computation is load balanced, and the number of processors is small enough that each processor has a significant amount of local computation to carry out. In our experiments, the use of null events increased the ratio of local to global computation a hundred-fold. A null event gives a lower bound on the time of the next event; the tighter the bound, the better the ratio of local to global computation.

Better speedup was obtained for preemptive-priority networks than for first-come-first-served networks. This may seem counter-intuitive, because the future behavior of a queue in a preemptive-priority network is more dependent on the current behavior of other queues. (Jobs depart a first-come-first-served queue in the order in which they arrive, no matter what happens at other queues, whereas in a preemptive-priority queue, a low-priority job is preempted by the arrival of a high-priority job.) The speedup for preemptive-priority networks was obtained for two reasons: first, simulating preemptive-priority networks requires more computation, even in a sequential implementation; second, null events are used to determine the future behavior of preemptive-priority queues.

- **Algorithms tailored to the physical system**

  The speedup obtained in our experiments is due in part to the algorithms being tailored to the problem: simulate a preemptive-priority queueing network. We expect the algorithm to work well on problems that have structures that can be exploited — problems such as circuit simulation and simulations of classes of queueing networks. We expect the algorithm to work poorly on problems that have no structure —

problems in which any PP can communicate with any other PP.

In summary, it is possible to obtain significantly faster execution of simulations on concurrent computers by matching the problem and the algorithm to the computer.

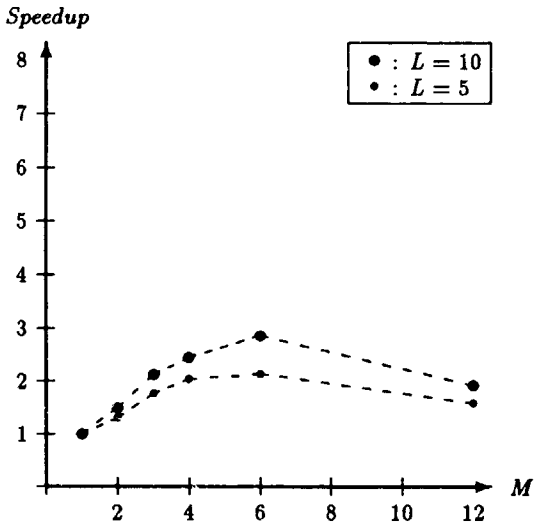


Figure 1 : $N = 12$, $J = 12 \times 10$

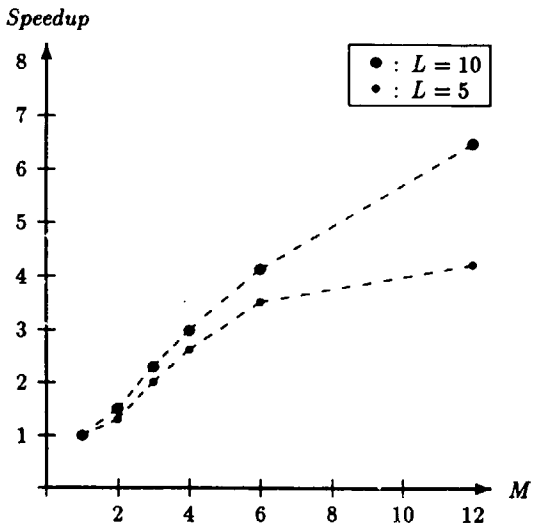


Figure 2 : $N = 12$, $J = 12 \times 50$

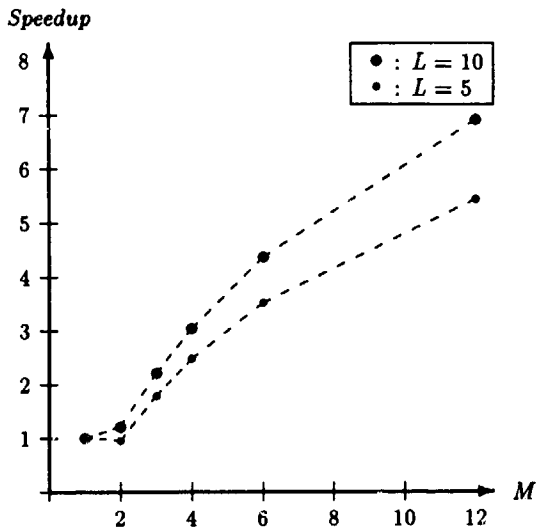


Figure 3 : $N = 12$, $J = 12 \times 100$

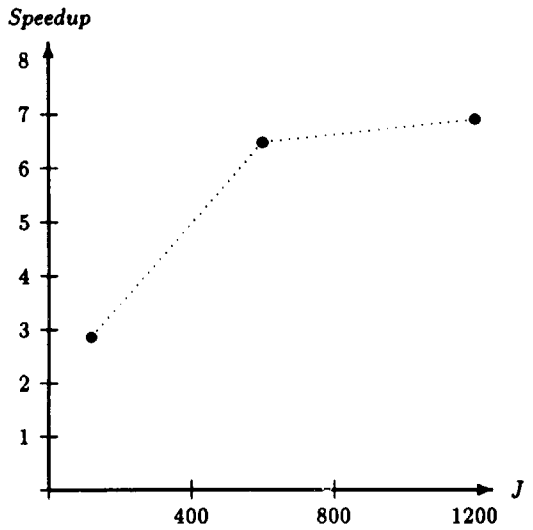


Figure 4 : Maximal speedup for $N = 12$

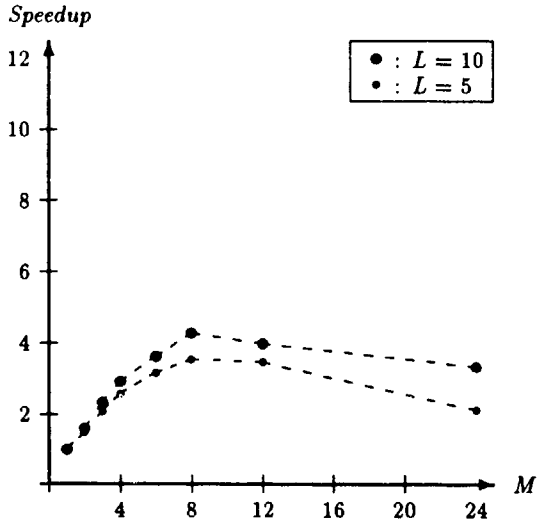


Figure 5 : $N = 24$, $J = 24 \times 10$

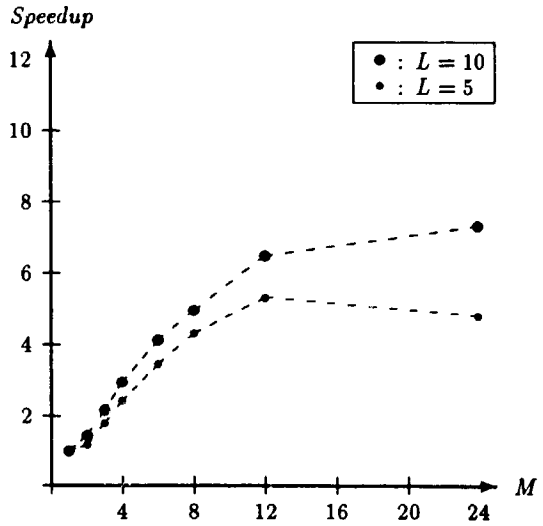


Figure 6 : $N = 24$, $J = 24 \times 50$

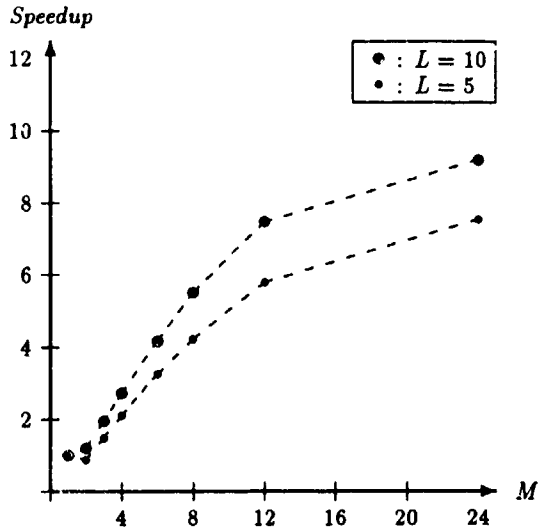


Figure 7: $N = 24$, $J = 24 \times 100$

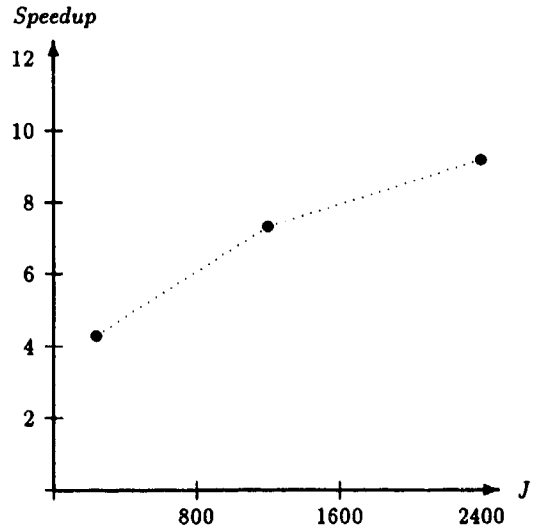


Figure 8 : Maximal speedup for $N = 24$

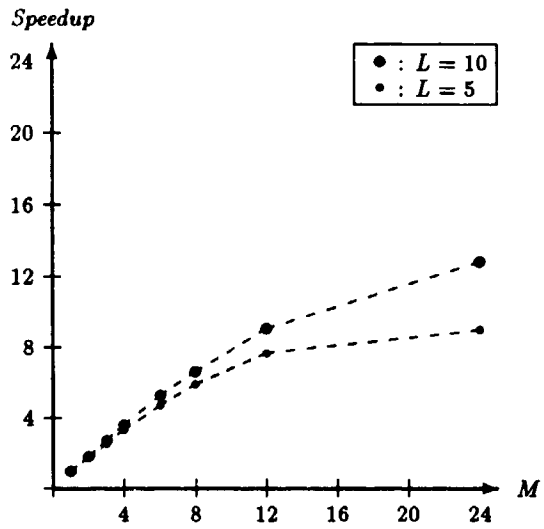


Figure 9 : $N = 24$, $J_h = 24 \times 10$, $J_l = 24 \times 10$

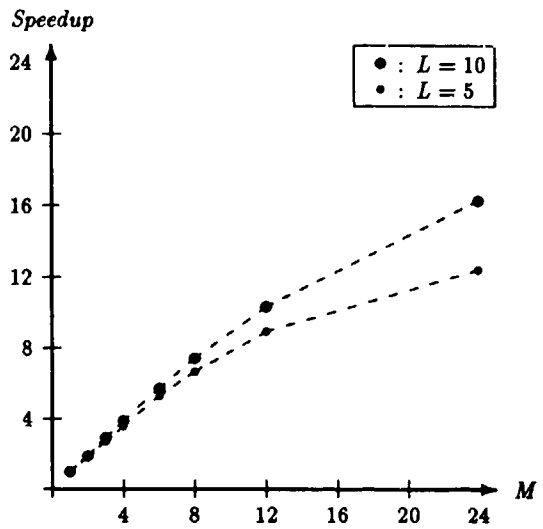


Figure 10 : $N = 24$, $J_h = 24 \times 10$, $J_l = 24 \times 50$

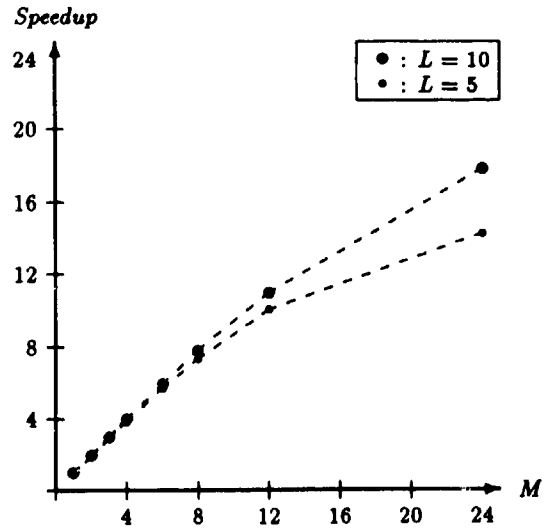


Figure 11 : $N = 24$, $J_h = 24 \times 10$, $J_l = 24 \times 90$

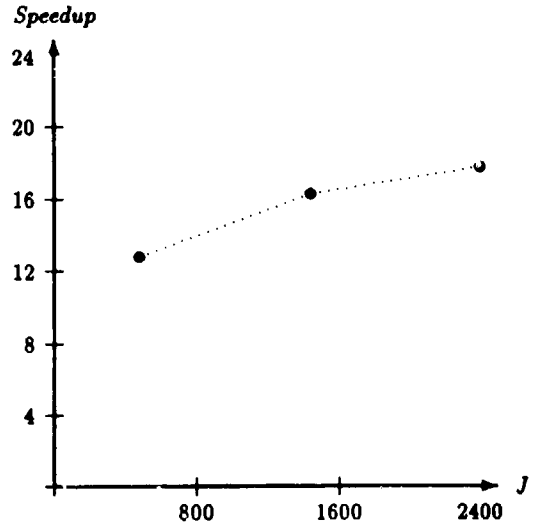


Figure 12 : Maximal speedup for priority queues, $N = 24$

References

[1] Bryant, R. E. *Simulation of Packet Communication Architecture Computer Systems*, MIT Laboratory for Computer Science, Technical Report TR-188, 1977.

[2] Chandy, K. M., and L. Lamport. "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Science*, **3:1**, February 1985, pp. 63-75.

[3] Chandy, K. M., and J. Misra. *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[4] Fujimoto, R. M. "Performance measurements of distributed simulation strategies," *Proceedings of the Conference on Distributed Simulation, 1987*, Simulation Series. Vol. 19, No. 3. The Society of Computer Simulation International, San Diego.

[5] Hartrum, T. C., and B. J. Donlan. "Distributed battle-management simulation on a Hypercube," *Proceedings of the Conference on Distributed Simulation, 1987*, Simulation Series. Vol. 19, No. 3. The Society of Computer Simulation International, San Diego.

[6] Jefferson, D. "Virtual time," *ACM Transactions on Programming Languages and Systems*, **7:3**, July 1985, pp. 404-425.

[7] Misra, J. "Distributed discrete-event simulation," *Computing Surveys*, **18:1**, March 1986, pp. 39-65.

[8] Reed, D. A., and A. D. Malony. "Parallel discrete event simulation: The Chandy - Misra approach," *Proceedings of the Conference on Distributed Simulation, 1987*, Simulation Series. Vol. 19, No. 3. The Society of Computer Simulation International, San Diego.

[9] Reynolds, P. F. "A shared resource algorithm for distributed simulation," *Proceedings of the Ninth International Computer Architecture Conference*, pp. 259-266, Austin, Texas, April 1982.

[10] Seitz, C. L., J. Seizovic, and W. Su. *The C Programmer's Abbreviated Guide to Multicomputer Programming*, Caltech-CS-TR-88-1, January 1988.

[11] Stone, H. *High Performance Computer Architecture*, Addison-Wesley, 1987.